

append vs cons vs list

This post isn't meant to be comprehensive. [Ask questions in lab or as a followup here if you're confused.](#)

One of its major flaws is that it doesn't cover box and pointers. **LEARN BOX AND POINTERS.**

Here is a beautiful web based Scheme interpreter that will draw box and pointer diagrams for you. Run through the examples below with this thing:

<http://xuanji.appspot.com/js-scheme-stk/index.html>

In order to understand these three procedures, you first have to understand a little about Pairs and Lists.

Pairs are data structures that have two slots. You can put different stuff in these slots, like numbers or words or sentences or booleans--pretty much anything. You make a pair using cons.

```
STk> (cons 'foo 'bar)
(foo . bar)
```

```
STk> (cons 1 'ring)
(1 . ring)
```

```
STk> (cons (+ 1 2 3) (member? 3 '(the 3 stooges)))
(6 . #t)
```

In order to get stuff from a pair that you have made, you use car and cdr. car gets the thing in the first slot. cdr gets the thing in the second slot.

```
STk> (define foo (cons 'x 'y))
foo
```

```
STk> foo
(x . y)
```

```
STk> (car foo)
x
```

```
STk> (cdr foo)
y
```

That was straight forward. Now for the trippy part:

You can put pairs inside of pairs:

```
STk> (define foo (cons (cons 3 4) 5))  
foo
```

```
STk> foo  
((3 . 4) . 5)
```

```
STk> (car foo)  
(3 . 4)
```

```
STk> (car (car foo))  
3
```

```
STk> (caar foo) ; functionally equivalent as above.  
3
```

```
STk> (cdr foo)  
5
```

```
STk> (cdr (car foo))  
4
```

```
STk> (cdar foo) ; functionally equivalent as above.  
4
```

There's a certain style of pair nesting that is especially useful--**Lists**.

Each list has these properties:

1. Every list is a pair **or** the empty list (denoted by '()' or nil).
2. The car of a nonempty list is some item.
3. The cdr of a nonempty list **must** be another list.

```
STk> (cons 1 (cons 2 (cons 3 '()))) ; list of numbers  
(1 2 3)
```

```
STk> (define stooges (cons 'larry (cons 'curly (cons 'moe nil))))  
stooges
```

```
STk> stooges
```

```
(larry curly moe)
```

```
STk> (car stooges)
```

```
larry
```

```
STk> (cdr stooges) ; Calling cdr on a non-empty list gives you another list!
```

```
(curly moe)
```

```
STk> (cadr stooges)
```

```
curly
```

```
STk> (cdar stooges) ; Why does this break?
```

```
*** Error:
```

```
cdar: bad list: (moe larry curly)
```

```
Current eval stack:
```

```
0 (cdar stooges)
```

```
STk> (define not-a-list (cons 'foo (cons 'bar 'baz))) ; This is not a list.
```

```
not-a-list
```

```
STk> not-a-list ; What property does this break?
```

Notice how Scheme knew that we were making lists. Before we had parens and periods which organized our items. Scheme now recognizes that we're making a list and does away with the periods and some of the parens.

If you stare a bit at the list rules above, you can notice we used a recursive definition to define lists. Recursion... **on data!**

.

Let's talk about list. List takes a bunch of stuff and makes a list out of them. The stuff can be anything. Words, numbers, pairs, other lists. list doesn't care.

```
STk> (list 'foo 'bar 'baz) ; Lists takes anything and makes a list out of it.
```

```
(foo bar baz)
```

```
STk> (list 'foo ((lambda (x) (+ x 4)) 8) #f (cons 1 (cons 3 4)) (cons 1 (cons 2  
nil)) (list 1 2 3)) ; ANYTHING
```

```
(foo 12 #f (1 3 . 4) (1 2) (1 2 3))
```

```
STk> (list 'x 'y 'z)
(x y z)
```

```
STk> '(x y z) ; Sometimes you can get away with using quote to make literal lists.
Yes, sentences are secretly lists.
(x y z)
```

Now we can talk about append:

```
STk> (append '(a b c) '(d e f) '(g h i)) ; Append takes in lists and appends them
together.
(a b c d e f g h i)
```

```
STk> (append 'foo '(1 2 3)) ; foo is not a list. Stuff will break.
*** Error:
      append: argument is not a list: foo
Current eval stack:
```

```
0    (append (quote foo) (quote (1 2 3)))
```

You know that cons makes a pair. You also know that you can make a list out of pairs. You can abuse cons for your own maniacal purposes.

```
STk> (cons 'joe stooges) ; Put stuff at the beginning of a list!
(joe larry curly moe)
```

The following only applies to the STk interpreter.

```
STk> (append '(1 2 3) 'foo) ; Wait... what?
(1 2 3 . foo)
```

```
STk> (append '(1 2 3) (cons 4 5)) ; The plot thickens!
(1 2 3 4 . 5)
```

```
STk> (append stooges 'shemp) ; You should really figure out why this works.
(larry curly moe . shemp)
```

To summarize:

append takes in lists and outputs a big list.

cons takes in things and makes a pair out of them. However, we know that lists are made of pairs, so we can throw together a list if we use cons a certain way

list takes in things and makes a list out of those things, regardless of what they are.

Illustrated summary: <http://csillustrated.berkeley.edu/PDFs/posters/list-constructors-1-poster.pdf>

Let me know if you have any questions.

Andrew